МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МО ЭВМ

ОТЧЕТ по лабораторной работе№2

по дисциплине «Алгоритмы и структуры данных»

Тема: Рекурсивная обработка иерархических списков

Вариант 23

Студент гр. 8304 Чешуин Д. И.

Преподаватель Фирсов М.А.

Санкт-Петербург

2019

Цель работы.

Познакомиться с нелинейной конструкцией – иерархический список, способами её организации и рекурсивной обработки. Получить навыки решения задач обработки иерархических списков, с использованием базовых функций их рекурсивной обработки.

Постановка задачи.

1) проанализировать полученное задание, выделив рекурсивно определяемые информационные объекты и (или) действия;
2) разработать программу, для решения поставленного задания, использующую рекурсию;
3) сопоставить рекурсивное решение с итеративным решением задачи;
4) сделать вывод о целесообразности и эффективности рекурсивного решения данной задачи.

Пусть алгебраическое выражение представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в постфиксной форме (<аргументы> <операция>)). Аргументов может быть 1 или 2. Представлены операции +, -, * и функции sin(), cos(), необходимо сделать упрощение.

Описание алгоритма.

Программа рекурсивно считывает данные и заносит их в список. Для этого считывается очередное слово. Если это атом, он добавляется к иерархическому списку, если это список – включается в состав создаваемого списка.

Для реализации разворота упрощения списка он сначала проверяется на корректный синтаксис, а затем в цикле рекурсивно упрощаются сначала все вложенные списки, после чего упрощается текущее выражение.

```
упростить (список)
{
        если аргумент1 == список
```

```
        новый аргумент1  = упростить(аргумент1)
    иначе
        новый аргумент1  =  аргумент1
    если аргумент2 == список
        новый аргумент2 = упростить(аргумент2)
    иначе
        новый аргумент2  =  аргумент2

    если сущ. правило упрощения(новый аргумент1, новый аргумент 2)
        новый список = упростить в соответствии с правилом
    иначе
        новый список = копировать(список)
    вернуть новый список
}
```

Спецификация программы.

Программа предназначена для упрощения иерархического списка.

Программа написана на языке C++. Входными данными являются символы английского алфавита, числа, скобки и знаки +,-,*. Они считываются из консоли или из файла. Выходными данными являются промежуточные значения вычисления выражения и конечный результат. Данные выводятся в консоль. Результат выводится в консоль либо в файл с именем – имя входного файла - result.

Тестирование.

```
a (b (b 1 *) -) +
b 1 * - is simplified to - b
b (b 1 *) - - is simplified to - 0
a (b (b 1 *) -) + - is simplified to - a

a (b (b 1 *) -) + |is equal to| a
```

Рисунок 1- Упрощение выражения

Остальные тесты приведены рисунках 2 и 3.

```
1 |is equal to| 1
a b + |is equal to| a b +
a 0 * |is equal to| 0
0 a * |is equal to| 0
1 a * |is equal to| a
a 1 * |is equal to| a
a 0 + |is equal to| a
0 a + |is equal to| a
a 0 - |is equal to| a
0 a - |is equal to| 0 a -
a a - |is equal to| 0
sin (PI) |is equal to| 0
sin (0) |is equal to| 0
cos (PI) |is equal to| -1
cos (0) |is equal to| 1
+ |is equal to| +
a + |is equal to| a +
a a a |is equal to| a a a
a a a + |is equal to| a a a +
```

Рисунок 2- Простые тесты

```
(a (a a -) +) ((cos (0)) (sin (PI)) -) * |is equal to| a
a (0 (c ((cos (0)) (sin (a)) *) +) +) * |is equal to| a (c (sin a) +) *
```

Рисунок 3 – Продвинутые тесты

Анализ алгоритма.

Алгоритм работает за линейное время от размера списка. Недостаток рекурсивного алгоритма – ограниченный стек вызовов функций, что в свою очередь накладывает ограничение на количество вложенных списков, а также затраты производительности на вызов функций.

Описание функций и СД.

Класс-реализация иерархического списка(List) содержит умный указатель на головной узел списка(Head) и умный указатель на хвостовой узел списка(Tail). Узел списка(Node) содержит умный указатель на следующий элемент списка, слабый – на предыдущий элемент списка. И умный указатель на значение(Data). Классы Atom и List отнаследованы от Data и используются в узлах. Умные указатели были использованы во избежание утечек памяти.

Методы класса List

```
void pushBack(Data::DataP data);
```

Помещается в список переданный элемент.

```
Data::DataP pullHead();
```

Извлекает головной элемент списка.

```
bool isEmpty();
```

Проверяет пустой список или нет.

```
Node::NodeP begin();
```

Возвращает указатель на головной элемент списка.

```
Node::NodeP end();
```

Возвращает указатель на последний элемент списка.

```
std::string toString();
```

Конвертирует список в строку.

Методы класса Parser

```
List::ListP parse(std::istream& stream);
```

Считывает строку из переданного потока ввода и создаёт из неё список.

```
Atom::AtomP readAtom(std::istream& stream);
```

Считывает слово из переданного потока ввода и создаёт атом.

Методы класса ListHandler

```
bool isValid(List& list);
```

Рекурсивно проверяет структурную корректность списка.

```
List::ListP simplify(List& list);
```

Создаёт новый экземпляр списка, после чего рекурсивно упрощает переданный список, записывая упрощённый список в новый экземпляр.

Структура списка (a (b (b 1 *) -) +) изображена на рисунке 4

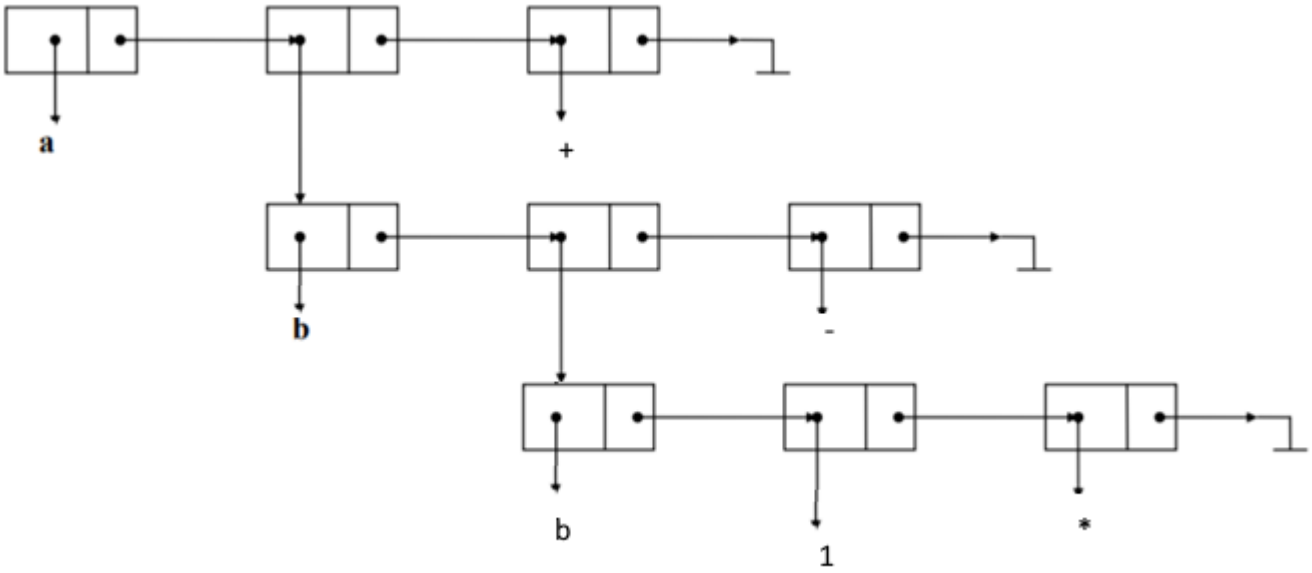

Рисунок 4 – Структура иерархического списка

Вывод.

В ходе выполнения данной лабораторной работы реализован класс иерархического списка и функция для его упрощения. Научился обрабатывать иерархические списки с использованием базовых функций их рекурсивной обработки и упрощать алгебраические выражения программным путём. Иерархический список реализован с помощью наследования самого списка и хранимого значения от общего предка.

Приложение А. Исходный код программы.

list.h

```
#ifndef LIST_H
#define LIST_H

#include <memory>
#include "data.h"
#include "node.h"
#include "atom.h"

class List: public Data
{
private:
    Node::NodeP head_  = nullptr;
    Node::NodeP tail_  = nullptr;
public:
    typedef std::shared_ptr<List> ListP;

    List();

    void pushBack(Data::DataP data);
    Data::DataP pullHead();
    bool isEmpty();

    Node::NodeP begin();
    Node::NodeP end();

    std::string toString();
};

#endif // LIST_H
```

list.cpp

```
#include "list.h"

List::List()
{
    this->setDataType(DataType::LIST);
}

void List::pushBack(Data::DataP data)
{
    Node::NodeP newNode(new Node);
    newNode->setData(data);

    if(head_ == nullptr)
    {
        head_ = newNode;
        tail_ = newNode;
    }
```

```
    else
    {
        tail_->setNext(newNode);
        newNode->setPrev(tail_);
        tail_ = newNode;
    }
}

Node::NodeP List::begin()
{
    return head_;
}

Node::NodeP List::end()
{
    return tail_;
}

std::string List::toString()
{
    std::string outString;
    Node::NodeP curNode = head_;
    List::ListP curList = nullptr;
    Atom::AtomP curAtom = nullptr;

    if(isEmpty())
    {
        return "empty expr";
    }

    while(curNode != nullptr)
    {
        if(curNode->data()->dataType() == DataType::ATOM)
        {
            curAtom = std::static_pointer_cast<Atom>(curNode->data());
            outString += curAtom->value();
        }
        else if(curNode->data()->dataType() == DataType::LIST)
        {
            curList = std::static_pointer_cast<List>(curNode->data());
            outString += '(' + curList->toString() + ')';
        }

        if(curNode->next() != nullptr)
        {
            outString += " ";
        }

        curNode = curNode->next();
    }

    return outString;
}
```

```
Data::DataP List::pullHead()
{
    DataP data = head_->data();
    Node::NodeP buf = head_->next();

    head_->setData(nullptr);

    head_ = buf;

    return data;
}

bool List::isEmpty()
{
    return head_ == nullptr;
}
```

listhandler.h

```
#ifndef LISTHANDLER_H
#define LISTHANDLER_H

#include <iostream>
#include <memory>
#include <map>
#include <string>
#include "list.h"
#include "atom.h"

class ListHandler
{
private:
    typedef std::shared_ptr<ListHandler> ListHandlerP;

    bool isArgument(Atom& atom);
    List::ListP simplifyFunc(std::string& func, Data::DataP arg);
    List::ListP simplifyOper(std::string& oper, Data::DataP arg1,
Data::DataP arg2);
public:
    ListHandler() = default;
    bool isValid(List& list);
    List::ListP simplify(List& list);
};

#endif // LISTHANDLER_H
```

listhandler.cpp

```
#include "listhandler.h"

bool ListHandler::isValid(List& list)
{
    if(list.isEmpty())
    {
```

```
    return false;
}

Node::NodeP buf = list.begin();
AtomType exprType = AtomType::UNKNOWN;

// Проверка 1 аргумента
if(buf->data()->dataType() == DataType::ATOM)
{
    Atom::AtomP arg1 = std::static_pointer_cast<Atom>(buf->data());

    if(isArgument(*arg1))
    {
        exprType = AtomType::OPERATOR;
    }
    else if(arg1->type() == AtomType::FUNCTION)
    {
        exprType = AtomType::FUNCTION;
    }
    else
    {
        return false;
    }
}
else
{
    List::ListP arg1 = std::static_pointer_cast<List>(buf->data());

    if(isValid(*arg1))
    {
        exprType = AtomType::OPERATOR;
    }
    else
    {
        return false;
    }
}

// Проверка 2 аргумента
buf = buf->next();

if(buf == nullptr)
{
    if(exprType == AtomType::FUNCTION)
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

```
if(buf->data()->dataType() == DataType::ATOM)
{
    Atom::AtomP arg2 = std::static_pointer_cast<Atom>(buf->data());

    if(isArgument(*arg2) == false)
    {
        return false;
    }

}
else
{
    List::ListP arg2 = std::static_pointer_cast<List>(buf->data());

    if(isValid(*arg2) == false)
    {
        return false;
    }
}

// Проверка оператора
buf = buf->next();

if(buf == nullptr)
{
    if(exprType == AtomType::FUNCTION)
    {
        return true;
    }
    else
    {
        return false;
    }
}

if(buf->data()->dataType() == DataType::ATOM)
{
    Atom::AtomP oper = std::static_pointer_cast<Atom>(buf->data());

    if(oper->type() != AtomType::OPERATOR)
    {
        return false;
    }
}
else
{
    return false;
}

//проверка законченности
buf = buf->next();

if(buf != nullptr)
```

```
    {
        return false;
    }

    return true;
}

bool ListHandler::isArgument(Atom& atom)
{
    return (atom.type() == AtomType::VARIABLE ||
            atom.type() == AtomType::CONST_VALUE);
}

List::ListP ListHandler::simplify(List& list)
{
    if(isValid(list))
    {
        if(list.begin()->next() == nullptr)
        {
            if(list.begin()->data()->dataType() == DataType::LIST)
            {
                List::ListP buf =
std::static_pointer_cast<List>(list.begin()->data());

                List::ListP newList = simplify(*buf);

                std::cout << list.toString() << " - is simplified to - ";
                std::cout << newList->toString() <<std::endl;
                return newList;
            }
            else
            {

                List::ListP newList(new List());

                Atom::AtomP buf =
std::static_pointer_cast<Atom>(list.begin()->data());
                Atom::AtomP newAtom(new Atom(*buf));

                newList->pushBack(newAtom);

                std::cout << list.toString() << " - is simplified to - ";
                std::cout << newList->toString() << std::endl;
                return newList;
            }
        }

        Atom::AtomP buf = std::static_pointer_cast<Atom>(list.end()-
>data());

        if(buf->type() == AtomType::OPERATOR)
        {
            std::string operStr = buf->value();
```

```
            Data::DataP arg1 = list.begin()->data();
            Data::DataP arg2 = list.begin()->next()->data();

            List::ListP newList = simplifyOper(operStr, arg1, arg2);

            std::cout << list.toString() << " - is simplified to - ";
            std::cout << newList->toString() <<std::endl;
            return newList;
        }
        else
        {
            Atom::AtomP func = std::static_pointer_cast<Atom>(list.begin()-
>data());
            std::string funcStr = func->value();
            Data::DataP arg = list.begin()->next()->data();

            List::ListP newList = simplifyFunc(funcStr, arg);

            std::cout << list.toString() << " - is simplified to - ";
            std::cout << newList->toString() << std::endl;
            return newList;
        }
    }
    else
    {
        std::cout << list.toString() << " - not simplified because
incorrect. ";

        return std::make_shared<List>(list);
    }
}

List::ListP ListHandler::simplifyFunc(std::string& func, Data::DataP arg)
{
    List::ListP list(new List());
    Data::DataP newArg = nullptr;
    //упрощение аргумента
    if(arg->dataType() == DataType::LIST)
    {
        List::ListP argExpr = std::static_pointer_cast<List>(arg);
        List::ListP simplifyedArgExpr = simplify(*argExpr);

        if(simplifyedArgExpr->begin()->next() == nullptr)
        {
            newArg = simplifyedArgExpr->pullHead();
        }
        else
        {
            newArg = simplifyedArgExpr;
        }
    }
    else
    {
```

```
    newArg = arg;
}

if(newArg->dataType() == DataType::ATOM)
{
    Atom::AtomP argAtom = std::static_pointer_cast<Atom>(newArg);
    //упрощения sin
    if(func == "sin")
    {
        if(argAtom->value() == "PI" || argAtom->value() == "0")
        {
            Atom::AtomP newArgAtom(new Atom("0"));
            list->pushBack(newArgAtom);
        }
        else
        {
            Atom::AtomP newFuncAtom(new Atom(func));
            Atom::AtomP newArgAtom(new Atom(*argAtom));
            list->pushBack(newFuncAtom);
            list->pushBack(newArgAtom);
        }
    }
    //упрощения cos
    else
    {
        if(argAtom->value() == "PI")
        {
            Atom::AtomP newArgAtom(new Atom("-1"));
            list->pushBack(newArgAtom);
        }
        else if(argAtom->value() == "0")
        {
            Atom::AtomP newArgAtom(new Atom("1"));
            list->pushBack(newArgAtom);
        }
        else
        {
            Atom::AtomP newFuncAtom(new Atom(func));
            Atom::AtomP newArgAtom(new Atom(*argAtom));
            list->pushBack(newFuncAtom);
            list->pushBack(newArgAtom);
        }
    }
}
else
{
    Atom::AtomP newFuncAtom(new Atom(func));

    list->pushBack(newFuncAtom);
    list->pushBack(newArg);
}

return list;
```

```
}

List::ListP ListHandler::simplifyOper(std::string& oper, Data::DataP arg1,
Data::DataP arg2)
{
    List::ListP list(new List());
    Data::DataP newArg1 = nullptr;
    Data::DataP newArg2 = nullptr;

    bool isNewArg1Simple = false;
    bool isNewArg2Simple = false;
    bool isAlreadySimplified = false;
    //упрощение 1 аргумента
    if(arg1->dataType() == DataType::LIST)
    {
        List::ListP arg1Expr = std::static_pointer_cast<List>(arg1);
        List::ListP simplifyedArg1Expr = simplify(*arg1Expr);

        if(simplifyedArg1Expr->begin()->next() == nullptr)
        {
            newArg1 = simplifyedArg1Expr->pullHead();
        }
        else
        {
            newArg1 = simplifyedArg1Expr;
        }
    }
    else
    {
        newArg1 = arg1;
    }
    //упрощение 2 аргумента
    if(arg2->dataType() == DataType::LIST)
    {
        List::ListP arg2Expr = std::static_pointer_cast<List>(arg2);
        List::ListP simplifyedArg2Expr = simplify(*arg2Expr);

        if(simplifyedArg2Expr->begin()->next() == nullptr)
        {
            newArg2 = simplifyedArg2Expr->pullHead();
        }
        else
        {
            newArg2 = simplifyedArg2Expr;
        }
    }
    else
    {
        newArg2 = arg2;
    }

    if(newArg1->dataType() == DataType::ATOM)
    {
```

```
    isNewArg1Simple = true;
}
if(newArg2->dataType() == DataType::ATOM)
{
    isNewArg2Simple = true;
}
//упрощение текущей операции
if(oper == "*")
{
    if(isNewArg1Simple)
    {
        Atom::AtomP buf = std::static_pointer_cast<Atom>(newArg1);
        if(buf->value() == "0")
        {
            Atom::AtomP newAtom(new Atom("0"));
            list->pushBack(newAtom);

            isAlreadySimplified = true;
        }
        else if(buf->value() == "1")
        {
            list->pushBack(newArg2);

            isAlreadySimplified = true;
        }
    }
    if(isNewArg2Simple && isAlreadySimplified == false)
    {
        Atom::AtomP buf = std::static_pointer_cast<Atom>(newArg2);
        if(buf->value() == "0")
        {
            Atom::AtomP newAtom(new Atom("0"));
            list->pushBack(newAtom);

            isAlreadySimplified = true;
        }
        else if(buf->value() == "1")
        {
            list->pushBack(newArg1);

            isAlreadySimplified = true;
        }
    }
    if(isAlreadySimplified == false)
    {
        Atom::AtomP newOperAtom(new Atom(oper));

        list->pushBack(newArg1);
        list->pushBack(newArg2);
        list->pushBack(newOperAtom);
    }
}
else if(oper == "+" || oper == "-")
```

```
    {
        if(isNewArg1Simple)
        {
            Atom::AtomP buf = std::static_pointer_cast<Atom>(newArg1);
            if (oper == "-" && isNewArg2Simple)
            {
                Atom::AtomP buf2 = std::static_pointer_cast<Atom>(newArg2);
                if(buf->value() == buf2->value())
                {
                    Atom::AtomP newAtom(new Atom("0"));
                    list->pushBack(newAtom);

                    isAlreadySimplified = true;
                }
            }
            else if(oper == "+" && buf->value() == "0")
            {
                list->pushBack(newArg2);
                isAlreadySimplified = true;
            }
        }
        if(isNewArg2Simple && isAlreadySimplified == false)
        {
            Atom::AtomP buf = std::static_pointer_cast<Atom>(newArg2);
            if(buf->value() == "0")
            {
                list->pushBack(newArg1);
                isAlreadySimplified = true;
            }
        }
        if(isAlreadySimplified == false)
        {
            Atom::AtomP newOperAtom(new Atom(oper));

            list->pushBack(newArg1);
            list->pushBack(newArg2);
            list->pushBack(newOperAtom);
        }
    }

    return list;
}
```

main.cpp

```
#include <iostream>
#include "parser.h"
#include "listhandler.h"
#include "ioManager.h"

using namespace std;

int main(int argc, char** argv)
{
```

```
    List::ListP list;
    List::ListP simlifyedList;
    Parser parser;
    ListHandler listHandler;
    IoManager ioManager(argc, argv);

    istream* curStream = ioManager.nextStream();
    while(curStream)
    {
        list = parser.parse(*curStream);

        simlifyedList = listHandler.simplify(*list);

        std::cout << std::endl;

        std::string buf;

        buf = list->toString() + " |is equal to| " + simlifyedList-
>toString();

        ioManager.writeLine(buf);

        delete curStream;

        curStream = ioManager.nextStream();
    }

       std::cin.get();

    return 0;
}
```

node.h

```
#ifndef NODE_H
#define NODE_H

#include "data.h"
#include <memory>


class Node
{
public:
    typedef std::shared_ptr<Node> NodeP;
    typedef std::weak_ptr<Node> NodeWP;

    Node() = default;

    void setNext(NodeP node);
    NodeP next();

    void setPrev(NodeP node);
    NodeP prev();
```

```
    void setData(Data::DataP data);
    Data::DataP data();
private:
    NodeWP prev_;
    NodeP next_ = nullptr;
    Data::DataP data_ = nullptr;
};

#endif // NODE_H
```

node.cpp

```
#include "node.h"
#include "list.h"
#include "atom.h"

void Node::setNext(NodeP node)
{
    next_ = node;
}

Node::NodeP Node::next()
{
    return next_;
}

void Node::setPrev(NodeP node)
{
    prev_ = node;
}

Node::NodeP Node::prev()
{
    return Node::NodeP(prev_);
}

void Node::setData(Data::DataP data)
{
    data_ = data;
}

Data::DataP Node::data()
{
    return data_;
}
```

parser.h

```
#ifndef PARSER_H
#define PARSER_H

#include<iostream>
#include<istream>
#include<string>
```

```
#include "list.h"
#include "atom.h"

class Parser
{
private:

public:
    Parser() = default;

    List::ListP parse(std::istream& stream);
    Atom::AtomP readAtom(std::istream& stream);
};

#endif // PARSER_H
```

parser.cpp

```
#include "parser.h"

List::ListP Parser::parse(std::istream& stream)
{
    char buffer = '\0';
    List::ListP parsedList(new List());

    while(stream.peek() != EOF)
    {
        stream.get(buffer);

        if(buffer == '(')
        {
            parsedList->pushBack(parse(stream));
        }
        else if(buffer == ')')
        {
            break;
        }
        else if(buffer == ' ');
        else
        {
            stream.unget();

            parsedList->pushBack(readAtom(stream));
        }
    }

    return parsedList;
}

Atom::AtomP Parser::readAtom(std::istream& stream)
{
    char buffer = ' ';
    std::string atomValue = "";
```

```
    while(stream.peek() != EOF)
    {
        stream.get(buffer);

        if(buffer == '(' ||
           buffer == ')')
        {
            stream.unget();
            break;
        }
        else if(isspace(buffer))
        {
            break;
        }
        else
        {
            atomValue += buffer;
        }
    }

    Atom::AtomP atom(new Atom(atomValue));
    return atom;
}
```

types.h

```
#ifndef TYPES_H
#define TYPES_H

enum class DataType
{
    ATOM,
    LIST,
    UNKNOWN
};

enum class AtomType
{
    CONST_VALUE,
    VARIABLE,
    OPERATOR,
    FUNCTION,
    UNKNOWN
};

#endif // TYPES_H
```

atom.h

```
#ifndef ATOM_H
#define ATOM_H

#include <memory>
#include "data.h"
#include "types.h"
```

```
#include "string"

class Atom: public Data
{
private:
    AtomType type_ = AtomType::UNKNOWN;
    std::string valueStr_ = "";
public:
    typedef std::shared_ptr<Atom> AtomP;

    explicit Atom(std::string value);
    Atom(const Atom& atom);
    AtomType type();
    std::string value();
};

#endif // ATOM_H
```

atom.cpp

```
#include "atom.h"

Atom::Atom(std::string value)
{
    this->setDataType(DataType::ATOM);
    valueStr_ = value;

    if(value == "+" ||
       value == "-" ||
       value == "*")
    {
        type_ = AtomType::OPERATOR;
    }
    else if(value == "sin" ||
            value == "cos")
    {
        type_ = AtomType::FUNCTION;
    }
    else
    {
        bool isNumber = true;

        if(isdigit(value[0]) == 0 && value[0] != '-')
        {
            isNumber = false;
        }

        for(int i = 1; i < value.length(); i++)
        {
            if(isdigit(value[i]) == 0)
            {
                isNumber = false;
                break;
            }
```

```
        }

        if(isNumber || value == "PI")
        {
            type_ = AtomType::CONST_VALUE;
        }
        else
        {
            type_ = AtomType::VARIABLE;
        }
    }
}

Atom::Atom(const Atom& atom)
{
    setDataType(DataType::ATOM);
    valueStr_ = atom.valueStr_;
    type_ = atom.type_;
}

AtomType Atom::type()
{
    return type_;
}

std::string Atom::value()
{
    return valueStr_;
}
```

data.h

```
#ifndef DATA_H
#define DATA_H

#include <memory>
#include "types.h"

class Data
{
private:
    DataType dataType_ = DataType::UNKNOWN;
protected:
    void setDataType(DataType type);
public:
    typedef std::shared_ptr<Data> DataP;
    Data() = default;
    DataType dataType();
};

#endif // DATA_H
```

data.cpp

```
#include "data.h"
```

```
void Data::setDataType(DataType type)
{
    dataType_ = type;
}

DataType Data::dataType()
{
    return dataType_;
}
```

ioManager.h

```
#ifndef CLIHANDLER_H
#define CLIHANDLER_H

#include <iostream>
#include <fstream>
#include <memory>
#include <sstream>

class IoManager
{
private:
    int argc_ = 0;
    char** argv_ = nullptr;
    int curArgNum_ = 1;

    std::istream* curInStream_ = nullptr;
    std::ostream* curOutStream_ = nullptr;

    void openNextStream();
public:
    typedef std::shared_ptr<IoManager> IoManagerP;

    IoManager(int argc, char** argv);
    ~IoManager();
    std::istream* nextStream();
    void writeLine(std::string line);
};

#endif // CLIHANDLER_H
```

ioManager.cpp

```
#include "ioManager.h"

IoManager::IoManager(int argc, char** argv)
{
    argc_ = argc;
    argv_ = argv;

    if(argc_ < 2)
    {
        curInStream_ = &std::cin;
```

```
            curOutStream_ = &std::cout;
        }
    }

    void IoManager::openNextStream()
    {
        if(curInStream_ == nullptr){
            curInStream_ = new std::ifstream();
            curOutStream_ = new std::ofstream();
        }

        if(curArgNum_ >= argc_)
        {
            if(curInStream_ != &std::cin)
            {
                delete curInStream_;
                delete curOutStream_;
            }

            curInStream_ = nullptr;
            curOutStream_ = nullptr;

            return;
        }

        std::ifstream* inFileStream =
static_cast<std::ifstream*>(curInStream_);
        if(inFileStream->is_open())
        {
            inFileStream->close();
        }
        std::ofstream* outFileStream =
static_cast<std::ofstream*>(curOutStream_);
        if(outFileStream->is_open())
        {
            outFileStream->close();
        }

        while(curArgNum_ < argc_ && !inFileStream->is_open())
        {
            std::cout << "Try to open file - ";
            std::cout << argv_[curArgNum_] << std::endl;

            inFileStream->open(argv_[curArgNum_]);

            if(inFileStream->is_open())
            {
                std::string outFile(argv_[curArgNum_]);
                outFile += " - results";
                outFileStream->open(outFile);

                std::cout << "File opened." << std::endl << std::endl;
            }
```

```
                else
                {
                    std::cout << "Can't open - file not founded" << std::endl <<
std::endl;
                }

                curArgNum_ += 1;
            }
        }

        std::istream* IoManager::nextStream()
        {
            if(curInStream_ == nullptr)
            {
                openNextStream();

                if(curInStream_ == nullptr)
                {
                    return nullptr;
                }
            }

            while(curInStream_->peek() == EOF)
            {
                openNextStream();

                if(curInStream_ == nullptr)
                {
                    return nullptr;
                }
            }

            std::string buffer;

            std::getline(*curInStream_, buffer);
            if(buffer == "")
            {
                return nullptr;
            }

            std::stringstream* sstream = new std::stringstream();
            *sstream << buffer;

            return sstream;
        }

        void IoManager::writeLine(std::string line)
        {
            *curOutStream_ << line << std::endl;
        }

        IoManager::~IoManager()
        {
```

```
    if(curInStream_ != nullptr && curInStream_ != &std::cin)
    {
        delete  curInStream_;
        delete  curOutStream_;
    }
}
```